2

CECOM

CENTER FOR SOFTWARE ENGINEERING

ADVANCED SOFTWARE TECHNOLOGY

Subject: Final Report - Issues Involved in Developing Ada Real-Time Systems

CIN: C02 092LA 0002

15 FEBRUARY 1989

Issues Involved in Developing Real-Time Ada Systems

by

T. P. Baker
Department of Computer Science
Florida State University
Tallahassee, FL 32306-4019

for

U.S. Army HQ
Center for Software Engineering
Advanced Software Technology
Fort Monmouth, NJ 07703-5000

9 October 1988

The views, opinions, and/or findings contained in this report are those of the author and should not be construed as a~ official Department of the Army position, policy, or decision, unless so designated by other documentation.

Contents

Acknowledgements

The preparation of this report was funded by the U.S. Army Communications Command (CECOM), through the U.S. Army Research Office and Battelle Laboratories. It draws heavily on background work supported by the Florida State University, the Boeing Commercial Airplane Company, the Boeing Aerospace Company, and the University of Washington. This especially includes development of the prototype Corset Ada runtime system, which was the basis for some experiments described in this report.

Ideas expressed in this report owe much to discussions with the members of the Ada Runtime Environment Working Group (ARTEWG) of the Special Interest Group on Ada (SIGAda) of the Association for Computing Machinery (ACM), and participants at several workshops on real-time Ada issues attended by the author.

Summary

Some experienced developers of hard real-time embedded defense systems have complained about problems with the Ada programming language. These problems are real, but resourceful application builders and Ada language implementors are finding effective ways to work around them. Unfortunately, these solutions are based on special features of particular Ada compilation systems. They may involve subtle compiler-dependent application code, and expensive modifications to the compiler or Ada runtime system to fit a particular application. Whether we can solve the problems of programming real-time systems in Ada is therefore not so much an issue as whether we can provide economical, and preferably standard, Ada solutions to these problems.

Modification to the Ada standard is not a sufficient solution, especially in the short term. Despite the benefits of commonality, it is unwise to rush to lock in solutions to problems that are not yet well understood in standards as rigorous as the Ada language. Moreover, real-time systems typically have inherent hardware-dependencies. Thus, there appears to be a need for more capability to economically adapt Ada language implementations to meet the special needs of real-time applications, based on existing Ada compilation technology and within the current Ada standard. (CF)

Most of the features needed by real-time applications can be implemented within the Ada runtime system (RTS for short). This is fortunate, since producing, validating, and maintaining custom versions of an Ada compilers is very costly. Ada runtime systems are much smaller and simpler than compilers. If there is a clean and well-specified interface to important RTS modules, those modules may be customized without compiler modification.

The existence of well-defined interfaces is a key to RTS customization. This can be approached at several levels. One of these is represented by the Model Runtime System Interface [MRTSI], which specifies an interface between the code generated by a compiler and the RTS. If adopted by compiler vendors, this interface could encourage the development and more widespread availability of Ada runtime systems designed to specifically support embedded real-time applications. A second important level of interface is within the RTS, between modules that are likely to need application-specific tailoring and the rest of the RTS. A third level of interface that appears useful is between the application program and modules within the RTS that are of direct use to the application.

1 Introduction

Despite the strengths of the Ada programming language, as compared to assembly language or other existing high-order languages, some experienced developers of real-time embedded systems do not believe Ada meets their needs; they fear that using Ada may require sacrificing both productivity and quality. This situation is disturbing, especially in view of the increasing reliance being placed on software in critical defense systems, and the increasing complexity of this software. In this report, we describe some of the problems encountered using Ada for developing real-time systems, and attempt to describe possible solutions to some of these problems. We focus in particular on the ways in which the development of economical solutions to these problems can be encouraged through well-defined runtime system interface conventions.

Section 2 enumerates some general problem areas and issues related to using Ada for real-time embedded applications. In Section 3 we outline some approaches for dealing with these problems, including RTS extensions, and the potential benefits of standard interfaces. Section 4 we describe work we have done toward defining one important level of RTS interface, the Model Runtime System Interface (MRTSI). Part of this work has been development of a prototype MRTSI implementation, to help verify the interface's feasibility, which is also described in this section. (More details of this prototype are provided in Appendices A and B.) Section 5 summarizes the results of our studies.

2 Real-Time Ada Problems and Issues

The effort that produced the Ada language was originally justified on the grounds that existing high-order languages did not adequately meet the needs of real-time embedded systems. Now that Ada is a reality, we find that it has met most of those needs, left some others unmet, and also introduced some new problems.

2.1 Real-Time Software

Real-time computer applications are characterized by interaction with real-world (physical) events, over whose timing they have little or no control. It is therefore a requirement of the software that it control its own execution timing so as to synchronize with the real world. If the timing of the software does not meet certain "hard" constraints, the entire system of which the software is a part may fail. Where the likelihood and consequences of failure to meet a hard timing constraint are sufficiently serious, achieving correct timing becomes a central design goal.

Besides timing constraints, real-time systems typically have a number of important secondary characteristics, which derive from their typical function, which is monitoring and controlling physical processes. These characteristics include concurrency, high reliability and fault tolerance, resource constraints (memory, power, and weight), and low-level interactions between the software and application-specific hardware devices. Many newer real-time systems demonstrate additional characteristics, due to increasingly ambitious application goals. These include complexity, distribution over several processors, and very long lifetimes. The latter in turn implies the application must be extensible and adaptable to new processors and peripheral hardware.

Real-time software has traditionally been developed in a combination of assembly language and higher-order language (e.g. FORTRAN, PASCAL, JOVIAL, CMS2). The use of higher-order languages has been mostly limited to components for which most high order languages would be adequate – that is, computational algorithms. Very time-critical components, interfaces to hardware devices, and scheduling and resource management functions have been programmed in assembly language. The latter functions are typically isolated in an "executive", which may be partly standardized, but is ordinarily tuned to the specific requirements of each application.

There are problems with this way of developing software. Chief of these is that it requires a great deal of highly skilled labor. Also, because of the level of detail that must be mastered, it does not scale up well to larger projects. A third problem is that existing real-time programs tend to be "brittle"; they break if one attempts to modify them. Nevertheless, this traditional approach does work, and there are people who have experience with it.

2.2 Ada's Conflicting Philosophical Origins

Perhaps one of the problems with Ada as a real-time language is that the designers were out of touch with the practice of real-time programming. One gets this feeling by reading the Ada Rationale [Rat], the references it cites, and the other issues of *SIGPLAN Notices* around that time. The flavor of the prevalent philosophy among programming language enthusiasts might be conveyed by the following idealistic assertions, which were widely held at that time:

- The best way to design is *top-down*, working from well-defined *requirements*.
- The direction of progress is toward ever more use of *automation*, *standardization*, and *abstraction*.
- Reducing the use of assembly language will increase programmer productivity, reduce the level of skill required of programmers, and make it easier to modify programs correctly.
- Standardization on one high-order language will reduce dependence on specialized programmers, and reduce the need for new software by allowing more code to be re-used.
- Much of the detailed design work involved in producing software, including that concerned with scheduling to meet timing constraints, can be automated; such automation will result in better and cheaper systems.

Enlightened believers in these principles held them in a regard that was something between axiomatic and moralistic.

The requirement definition process for Ada brought in a wide variety of people, with different philosophies and experiences. These included experienced real-time programmers. Looking at the requirements this group produced, we can see the influence of pragmatic minds, who probably recognized that principles like those above do not apply quite so well to real-time software as they may in other contexts:

- Top-down design does not work well when requirements are not well-defined or subject to change, or when the designer does not have sufficient experience to choose decompositions that are efficiently implementable at lower levels.
- Automation only leads to savings when applied to essentially repetitive tasks and when the volume of production is sufficient to offset the cost of automation.
- Standard products frequently don't quite fit our requirements. (Consider, for example, shoes and feet.) Ideally, the illness of fit is offset by the economy of using the standard, but there are cases where insistence on using a standard that does not fit can be crippling.
- Abstraction, as traditionally applied to programming language semantics, throws away many details that are sometimes critical in real-time systems, such as the machine representation of data and exact execution timing.

Thus, while the Ada requiments specify a general-purpose high-order algorithmic language, they also clearly state that programmers should have the option of taking explicit control over critical low-level details. For example, the 1978 Ironman document [Iron] says "it shall be possible to use object machine features directly in programs". This and some other similar requirements were honored in the language design, and are traceable forward to features in present-day Ada. On the other hand, we see places where the idealists appear to have triumphed.

Examples of places where idealism appears to have overcome pragmatic considerations include the Ironman requirements that "The built-in mutual exclusion and synchronization facilities shall be sufficiently low-level to permit ... user definition of more specialized mechanisms," and "Within any parallel control structure it shall be possible to dynamically alter the relative priorities of the the paths." Such unmet requirements account for most of the present problems with Ada for real-time.

2.3 Early Problems with Ada

At first it seemed that Ada created more problems than it solved. There were no software tools available, no existing body of code that could be re-used, and no experienced programmers. The first Ada implementations were not very good, either.

The first Ada compilers were large, slow, unreliable, consumed a terrific amount of disk space, and were awkward to use. They produced code that was grossly inefficient, in terms of both size and speed. These compilers were designed to execute and generate code for main-frame computers running general-purpose operating systems. Because the low-level programming features of Ada were optional, and not needed in this environment, they were not implemented. For similar reasons, tasking operations were very slow. Because these compilers were targeted to virtual memory machines, they made no attempt to avoid linking useless runtime system components. The program images were therefore frighteningly large.

The language-lawyers and bureaucrats did not help either. They constantly harped about validation and full compliance with the Ada standard down to its last wart and mole (but not the optional features needed by real-time programmers). This diverted effort from achieving better object-program performance, and discouraged efforts to correct flaws and shortcomings of the language. (As compared, for example, to the free evolution of Pascal and Modula2.)

Another problem was the "bundling" of Ada with huge integrated programming environments. The Ada implementation was viewed as a closed system, integrating all the functions traditionally provided by separate software components, including executive, input-output (I/O) system, linker, loader, editor, and software library manager. This seemed to raise the cost and risk of using Ada, since proven tools would have to be replaced by new untested ones. Worse, since compiler developers invented their own closely closely held interfaces between tools, if one of these components should not prove adequate as provided by the compiler vendor, there appeared to be no way for the customer to modify or replace it.

2.4 Ada Today

Fortunately, these problems are largely past. Many people have learned and are teaching Ada. The body of reusable Ada code is rapidly growing, as is the number of Ada software tools. Today's Ada compilers are much better. They are faster, partly due to compiler improvements and partly due to advances in low-cost computers and disk drives. The best Ada compilers generate code comparable in size and speed to that generated by compilers for other languages. Some are now supporting all of Ada's optional features, including in-line machine-code procedures. Linkers now discard unused code, and runtime systems are smaller and faster. The language bureaucrats have gotten their priorities straight, so that validation no longer is a serious obstacle to producing usable compilers. Even the problem of bundling has faded slightly, so that some of today's compilers follow object code conventions compatible with standard operating system tools, including compilers for other languages. In short, Ada has caught up with other languages.

Now that most of Ada's early developmental difficulties are over, we can more accurately evaluate its suitability for writing real-time software, and focus more clearly on solutions to its remaining problems. However, before turning to criticize Ada, we must be careful to define the standard against which we will measure it. The first standard we should apply is established by other available programming languages. As compared to these, we will see that Ada looks good. The second standard by which we must judge Ada is that established by existing real-time operating systems or executives (executives, for short). This may seem unfair, especially since Ada doesn't come out very well, but Ada forces the comparison by getting involved in concurrent programming. Finally, we can measure Ada against directly against the characteristic requirements of real-time systems, but remembering that we are comparing a reality against an ideal. We will see that Ada has gotten off to a good start, but still has a long way to go.

2.5 Ada *versus* other Languages

Overall, as compared with other available languages, Ada looks very good. It has the best-defined and best-enforced standard, with a stringent validation process. After a large investment, there are now many conforming implementations, including several for machines that are well-suited for real-time applications. Ada supports clean interfaces, through strong typing and packages. It is extensible, through packages, generics, attributes, and pragmas. Many of the implicit dependencies on specific compilers and computer hardware that make programs written in other languages difficult to port can be eliminated or controlled, using Ada's attributes and representation clauses.

On the negative side, Ada is more complex than other standard languages. This causes several problems. Chief of these is that compared with compilers for other languages, Ada compilers are still large, slow, and use lots of permanent file space. They are large and slow because Ada requires more compile-time checking, and also because Ada needs more optimization. They use lots of permanent file space because of the separate compilation library facility.

Ada's complexity is also reflected in the object code that compilers produce. There are several language features that require complex optimizations in order to achieve acceptable performance. These include nonstatic sized arrays and records, aggregates, recursion, runtime checks, and tasking. Because of the complexity of the compiler code that performs these optimizations, there is more chance of compiler bugs. There may also be performance surprises, when expected optimizations are not applied, and errors, when a compiler applies an optimization in a context where it is unsafe.

Several Ada features, including separate compilation, overloading, and static expression evaluation, make the automatic processing and transformation of Ada source code difficult. This makes software tools for Ada inherently more complex than for other languages, and therefore more expensive. Fortunately, this extra cost has not prevented the development of Ada tools.

Reliable formal verification of Ada programs is made more difficult for Ada than some other languages by its more complex features, especially exception handling and task abortion. It is also made unreliable by aspects of the Ada semantics that the Standard leaves unspecified, but this is also true of other languages. Moreover, the practicality of formal verification of whole programs is already questionable, regardless of language.

Bundling Ada compilers with other software components remains a problem, though it is perhaps not as severe as it once was. This problem is more or less inherent in the Ada language definition, which imposes specific requirements on the functions of the executive, I/O system, linker, loader, and software library, and their interactions. As a consequence, Ada is often criticized for problems that one would not ordinarily blame on a programming language.

There are several directions in which Ada reaches outside the traditional programming language domain. First, by specifying a model of concurrent programming, Ada intrudes into the domain of the executive. Examples of this intrusion include task priorities, delays, hardware interrupt entries, intertask data sharing, and notification of certain exceptions. Second, by requiring compile-time checking across all the separately compiled units of a program, Ada intrudes into the domains of the program library manager and linker. Ada compilations systems intrude further into the domain of the linker when they attempt optimizations that require global information. Finally, to a lesser extent, Ada intrudes into the I/O system through the standard I/O packages and the ability to bind hardware interrupts to task entries.

By reaching into these other domains, Ada and its implementations limit a programmer more than do other languages. These limitations can be viewed as flaws when they stand in the way of getting a job done. For example, if a system designer needs to use deadline-based task scheduling the Ada priority rules get in the way. Suppose the designer wishes to load portions of his program, including code and certain constant tables, into read-only-memory. Ada provides no way of doing this, so the linker integrated with the Ada compiler probably won't allow it. What if a programmer wants to update a running program, without restarting the system? ... These are just a few of the problems.

2.6 Ada *versus* an Executive

Because Ada intrudes into the domain of the real-time executive, by supporting features like concurrent execution, delays, and task priorities, it preempts the application programmer from installing his own executive. However, since the standard Ada language does em not provide all the services typically provided by such an executive, the application programmer is left without some needed functionality. An Ada implementor can choose to provide additional services, but has no recognized source of guidance as to what these additional services should be.

An example of this sort of problem is how to program a collection of periodic tasks such that if a task misses its deadline it will be suspended immediately, regardless of its priority. Another example is how to insure the interleaved execution of two equal-priority tasks without the possibility of processor idle time.

Besides appropriately general scheduling primitives, Ada is missing some important abstractions that are normally provided by an operating system or executive. Examples of such missing abstractions include the concept of interrupt (the Ada notion of an address is not adequate), interrupt mask, storage area, data segment, timer, loadable code module, recovery unit, and processing node. Other abstractions are provided in only a crippled form, for which needed operations are not available. These include tasks, exceptions, procedures, and addresses. For example, there is a standard attribute to obtain the address of a task, procedure, label, or object, but there is no standard way to convert an address back to one of these entities.

Ada implementations that are targeted to machines with standard full-function operating systems typically provide extensions that permit access to the full capabilities of the underlying system. However, fundamental semantic incompatibilities between the operating system's process model and the Ada task model may still force the programmer into compromises, such as discarding Ada tasking in favor of "multiprogramming". (In some cases this may be a good idea.)

2.7 Interaction with the Real World

For interaction with the the real world, Ada allows a programmer to bind interrupts to task entries. The Ada implementation may map some interrupts (more properly traps) to predefined exceptions, but the programmer cannot bind interrupts to exceptions. This model is helpful, but is not entirely adequate in several respects:

1. There is no way to disable or block interrupts, except perhaps implicitly via control structures within the handler task.
2. The standard way of naming interrupts, via a value of type `SYSTEM.ADDRESS`, is not applicable to most architectures.
3. There is no way to bind interrupts to different handlers dynamically, without creating and destroying tasks.
4. There is no way for a hardware interrupt handler to reliably wake up another task, that may be waiting. This appears possible using a rendezvous, but the implementation practicalities

of interrupt handlers require that such a rendezvous be conditional on the side of the handler, so that wakeup signals may be missed.

5. There is no way for an interrupt handler to preemptively and asynchronously signal an executing task to change its thread of control. This means, for example, that if an interrupt handler detects that another task is about to miss its deadline, or detects that an external event has occurred requires the task to change its priority and restart with new data, there is no way for the handler to force the task to take immediate notice.

6. There is no way to specify how an interrupt is to be treated when the task with the corresponding entry is not ready to accept. Specifically, is it is queued or is it lost?

7. There is no standard pragma like `SHARED` for array and record structures, to insure that if access to a buffer is shared between an Ada tasks and an I/O device or interrupt handler the compiler will insure that elements of the buffer are read from and written to memory immediately.

2.8 Achieving Correct Timing

Ada gives very little control over execution timing. Achieving correct timing requires several things:

1. accurate estimation of processor time and other resource requirements of processes;

2. an accurate real-time clock;

3. a means of preempting the processor for certain timed events;

4. control over the scheduling of all resources needed by time-critical processes.

Ada, like virtually all high level languages, provides no information about the execution time of code sequences. The problem is slightly aggravated by Ada's complexity. Ada aggravates this problem further by including dynamic storage allocation features that may cause an implementation to perform time-consuming storage allocation and reclamation operations at unpredicatable times.

Ada does provide a real-time clock (2) in the form of `CALENDAR.CLOCK`, but without any assurance of accuracy, and with no specification of how or when this clock may be reset, or whether it may occasionally jump backward. The range and precision of the type of value returned by this clock make it inefficient for most real-time uses, since it is too long to be read and written atomically on a machine with less than 64-bit operations.

Ada does not provide a preemptive timer, except possibly via machine-dependent linkage to an interrupt generated by a hardware timer. This is likely not to be readily available since in many cases the Ada RTS will already rely on having exclusive use of the only available hardware timer(s), for the implementation of `CALENDAR.CLOCK` and delays. The **delay** statement provides a form of relative time delay with no known accuracy, no guaranteed upper bound, and no specified relationship to `CALENDAR.CLOCK`. Because the reading of the clock, calculation of delays, and execution of the delay statement are all preemptable, there is no way of achieving an accurate absolute or periodic wakeup event. Some sort of virtual periodic and absolute timers therefore should be provided.

Static task priorities are provided, but they are optional, and crippled in several ways. First of all, they are assigned to task types, rather than task objects. Second, they are assigned statically, which causes difficulty adapting scheduling to changing real-world requirements (e.g., modes). Third, there are anomalies between the priority scheme and the semantics of entry calls[CSLRT]. For example, while the priority of a calling task is inherited by the accepting task, this does not take place until a rendezvous begins. Thus, if the accepting task is not ready to accept, a high priority caller and its acceptor may wait forever while middle-priority tasks execute. Similar anomalies can arise when the FIFO entry service policy causes a high priority task to be queued behind a low priority one, and when the incompletely specified semantics of selective waits allows a low priority call to be accepted ahead of a high priority one.

Ideally, Ada should provide the freedom (and means) to devise and implement application-specific scheduling policies. This means there should be at least a way to change priorities and to suspend and resume tasks. A standard priority-based scheme for consistent cpu-scheduling and rendezvous is also needed, with a guaranteed minimum range of priorities[CSLRT].

Ada also hides or takes away memory-related choices (as described below), that may affect timing in a hierarchical memory system.

2.9 Control over Memory Usage

Ada provides some control over memory usage. Specifically Ada has:

1. the STORAGE_SIZE representation clause for tasks;
2. the STORAGE_SIZE representation clause for access types;
3. the pragma CONTROLLED for access types;
4. the generic subprogram UNCHECKED_DEALLOCATION, for access objects;
5. other representation clauses for data types.

The first problem with these mechanisms is that they are optional, and in the case of pragmas can be ignored without comment. The second problem is that the effects of these features are imprecisely defined. For example, does the storage size of a task include storage for objects of access types declared within the task? How about storage for local tasks within it? How about other heap storage, for implicitly created temporary objects of dynamic size within the task? Is this included within the storage of the task? Can this storage be relied upon to be contiguous? How much of this storage is overhead? How efficient is the allocation and recovery mechanism; i.e., how much will be wasted due to fragmentation? Similar questions can be raised about storage for access objects.

The third problem is that there are many storage management issues not addressed by any Ada feature, no matter how imprecisely defined. For example: How can objects be assigned to read-only memory (ROM) versus random-access memory (RAM)? How about between shared multiprocessor memory and local memory? How can specific data be clustered within a single page, or aligned on page or segment boundaries? If there is virtual memory, how can specific objects or code be locked into physical memory? How can specified code or data be prefetched at a scheduled time, or locked into RAM, so that it will be resident when needed? How can certain sections of memory be protected against accidental overwriting? Control over such choices is often provided by traditional executives, and could be provided in Ada via appropriate extensions.

2.10 Concurrent Programming

The reasons for including concurrent programming (tasking) in a programming language are conve nience and commonality. By enforcing use of one standard model of concurrency, the language can make it easier for people to understand one another's programs and save time that might otherwise be spent on developing an application-specific executive. Were it not for these reasons, equivalent functionality could just as well be provided via service calls to an underlying operating system or executive (executive, for short). Moreover, if the interface to this executive is standardized, the goal of commonality is also met. The only good reason for including tasking as part of the Ada language is therefore the notational convenience of special syntactic forms.

An immediate negative consequence of embedding a model of concurrency into Ada is that it conflicts with other established models, including virtually all those used in the literature of scheduling theory and all those used by existing standard executives. Of course, this would be true of any attempt to impose a standard model of concurrency, including a standard executive interface, so if we accept the need for a standard model of concurrency this is not an issue.

The question remains whether Ada's model of concurrency is adequate. The answer to this question is "not entirely". The Ada tasking model appears to be adequate for some purposes, but inadequate for others, including specifically many real-time applications.

Ada supports concurrent programming, via the task mechanism, but this tasking mechanism has several flaws. The main problem is an inappropriate choice of primitives. They are inappropriate because they are too complex to be efficiently implemented, and because they need to be used in complex combinations to perform functions that are frequently needed in real-time systems. The semantics of task creation and termination are especially onerous. The rendezvous is also too complex to be the lowest-level method provided for task synchronization and mutual exclusion. For example, coding basic operations on standard concurrent programming abstractions such as buffers and monitors requires the introduction of intermediary tasks and multiple rendezvous. This is gross inefficiency, as is revealed by examination of the runtime system code that implements the Ada "primitives". This code itself makes use of more natural primitives, such as buffers, queues, and semaphores!

Of course, one can argue that traditional primitives, such as buffers and monitors, are too low-level – that one should design using rendezvous, eliminating the need for these other constructs. (This does not address some other big issues: data sharing between tasks; dynamic task creation and termination; abortion.) However, experience using Ada tasking for real-time programming has proven that this belief is unfounded. Rendezvous rarely is useful by itself. For instance, the main reason for separating a program into tasks is to deal with asynchronism in the real world. By forcing such tasks to communicate through rendezvous, they are forced to synchronize, defeating the purpose of the original separation. We are then forced to introduce another task, and another rendezvous, where what we really wanted was a simple buffer.

Another belief is that Ada compilers should recognize Ada "idioms" for frequently used concurrent programming abstractions, and translate them into simpler constructs. Thus, a task whose only role is to serve as a buffer can be eliminated and the rendezvous replaced by simpler read and write operations on the buffer. This, too, is a fallacy. First of all, the conditions under which this sort of "optimization" can be performed safely are difficult to recognize, and easy to violate accidentally through program modifications. Compilers (and programmers) are thus forced to choose between safety and performance. Second, portability is lost, since correct timing is now dependent on a compiler being able to translate a particular idiom. Finally, the Ada source code is still difficult to comprehend, for much the same reasons as control structures constructed out of **gotos** are harder to understand than standard control structures like **for** loops and **if...then...else** statements.

An obvious solution to this problem of missing primitives is to export truly primitive operations directly to the programmer. This can be done via predefined packages, which would start out as implementation-dependent, but eventually might be standardized. This is simple, direct, reliable, economical, and presents less of a portability problem than complex compiler optimizations. If the needed predefined packages are not supported by a particular compiler, they can be written by the application programmer (ideally, using Ada with machine-code inserts). In some cases this may require detailed information about the Ada tasking implementation, but this is less work than adding tasking optimizations to a compiler.

We have already discussed some other specific problems with the Ada tasking model, underInteraction with the Real World" and "Controlling Execution Timing". These include: priorities, which are presently treated inconsistently, and need to be made dynamic for some applications; scheduling support for periodic processes; an immediate task restart capability; a way to reliably and asynchronously forward events requiring an immediate response from interrupt handlers to other tasks. Major deficiencies in the tasking model which we have not discussed include the need for some way of identifying tasks of heterogeneous types via a single data type, as in the context of schedulers and other resource managers, and the need for a non-blocking form of message-passing between tasks.

2.11 Reliability

In many ways, Ada aids the production of reliable software, by strong type checking and by the package mechanism, which encourages clean separation of concerns. For situations where the standard Ada tasking model is adequate, it may contribute to reliability, by eliminating the need for the application programmer to be concerned with the complexity of programming an executive.

The main obstacle to reliability imposed by Ada is its complexity. This increases the likelihood of programmer errors as well as language implementation errors (as discussed in Section 2.5). There are several interesting examples of compiler errors that have cause intermittent faults, and programming errors that may not cause a fault until after a program has been running for a very long time. One example is when a program uses a task access type to create a short-lived task in response to an infrequent event. In some implementations, the small residual storage occupied by such tasks after they terminate can eventually exhaust storage, but not until a long time has passed.

A secondary problem is implementation-dependencies over which no mechanism is provided for explicit control. This is more likely to cause portability problems, but might also cause reliability problems when going to a new version of an Ada compilation system. For example, a change in the default processor-scheduling algorithm might cause a multitask program to stop working correctly.

2.12 Fault Tolerance

Ada does a little to help with fault-tolerant design, through the exception-handling mechanism. This has some weaknesses, however. Chief among these weaknesses is that no provision is made for one task (when it detects an error, such as a timing fault) to raise an exception asynchronously in another (to cause it to recover from the error).

2.13 Operating within Constrained Resources

The problem of designing an Ada program to operate within resource constraints has already been discussed. Part of the problem is a consequence of language complexity. Fortunately, Ada compilers are getting better at recognizing and optimizing the simple structures that make up the bulk of most programs. The rest of the problem is a consequence of lack of control over time and storage utilization, which are discussed under "Achieving Correct Timing" and "Control over Memory Usage".

2.14 Interactions with Hardware

Ada goes a long way toward providing means for specifying interactions with hardware. These include representations clauses for most data representation dependencies, machine-code inserts, and interrupt entries. However, there are still quite a few hardware dependencies for which no specification mechanism is provided.

We have already discussed problems in the realm of interrupts, under "Interaction with the Real World".

In the real of representation clauses, there is no specification mechanism for array layout, including direction and alignment of elements. There is no way of specifying object alignment (e.g., word, page, segment). There is no standard for machine-code inserts, or for references within such inserts to entities declared elsewhere in the Ada program. There is no way to get an access value that designates a statically allocated interface object, such as a buffer. There is no way of specifying an interface data object, and no way of specifying a separate linker-name for an interface subprogram. There is no way of specifying a function in machine-code. Many implementations do not permit specifying some important attributes, especially values `'small` attribute for fixed-point types that are not a power of 2. (This is especially important because the input and output data, as well as the timing periods, of many hardware devices are decimal fractions.)

2.15 Portability

Ada does go farther to help portability than any language up to this point. Anecdotal reports comparing experience porting programs in Ada versus other languages make Ada look very good. This is due in large part to Ada's provision for explicit specification of data representation and the precision of numeric types, type attributes, and especially the strict validation and subsetting policy. It should therefore not be criticized too much for still having a few problems.

As with any language, Ada's remaining portability problems are due to implementation dependencies allowed by the Standard. A few especially troublesome examples of these include: a few arithmetic operations that require the standard type INTEGER; differences in task implementations, such as processor scheduling and the accuracy of of the real-time clock; unimplemented optional machine-dependent features.

2.16 Parallel and Distributed Systems

Support for parallel processing is one area where Ada is very weak. The tasking model is fairly well adapted to a multiprocessor system with mostly shared memory. Ada does not adapt so well to more distributed multiprocessor configurations, due to the need to support access to shared memory between tasks. Of course, this can be virtualized, but the communication overhead is high. Similarly, the cost of rendezvous goes up quickly across a distributed system. Worse, there is no non-blocking alternative to rendezvous for intertask message passing. There are some conceptual problems with systems of heterogeneous processors, but these disappear if the software is decomposed into separate programs on each machine.

Like all present conventional programming languages, Ada is likely to be made obsolete by the machines that are now on the horizon. When it comes to vector machines, Ada is probably no better than FORTRAN. The overloading does make it convenient to define and use explicit vector operations, but automatic vectorization is likely to run into problems with exceptions. By the time we reach large highly-parallel networks of small processors, and neural nets, the Ada task model is altogether inappropriate. Other problems arise from several places where the Ada standard specifically rules out concurrent execution, by stating that execution takes place "in some order".

2.17 Defense of Ada

Having made so many criticisms, we should say in Ada's defense that one cannot realistically expect a standard programming language to provide solutions to all these problems, since most of them are peculiar to specific applications and processor architectures. Nevertheless, real-time application builders will need to be able to solve these problems using Ada. The can, but not using ready-made standard solutions. The best course may be for Ada users to evolve some standard solutions for the most frequently occurring problems, while recognizing that there will always be special cases that require special solutions.

3 Approaches to Using Ada for Real-Time

3.1 Living With the Present Situation

Ada is currently being used, with some success, for real-time programming. There are complaints, most of which we have discussed. Still, Ada seems to be at least as successful as other high level languages before it. Where Ada is not adequate, there is still recourse to assembly-language or machine-code inserts. It appears that Ada tasking is being abandoned for most time-critical applications, being replaced by more traditional executives. Where the cost can be justified, Ada language implementations are being tailored to provide needed features that are missing from the standard language.

3.2 Language Changes

There is talk about language changes. It is likely that the "Ada9x" process will help clear up some problems. For instance, there appears to be strong support for reintroducing some form of asynchronous exception mechanism, similar to the "**raise** T'failure" feature that was dropped during the 1983 language revision. It is also likely that inconsistencies within the present priority scheme will be eliminated. However, it would be unwise to depend on the language change process for any solutions to really pressing problems.

Language changes are not likely to be approved until some time in the early 1990's and then they will take more time to be implemented. They are therefore a long-term solution. Besides, even if we could change Ada and implement the changes immediately, it is not clear what changes are needed, or whether it is the language that needs changing at all.

Many fundamental problems of real-time software engineering are still not well solved. We can expect that new techniques will be developed in the coming years. For example, recent research in the areas of real-time scheduling (c.f., [LNL,LSS]) has shown that the simple static priority-driven processor allocation model of Ada is inadequate, and opened up a wide range of potential alternate scheduling techniques. Based on what is known, it appears that any small set of scheduling techniques we might choose to support as standard options today are likely to prove too limiting. For this reason, scheduling experts suggest that Ada should not specify any standard scheduling technique, but should leave this option to the application programmer. Similar uncertainty exists about some other hard problems of real-time systems, such as distribution, fault tolerance, and reconfiguration. These problems are still not well enough understood to admit to uniform standard solutions.

It is clear that we cannot expect a programming language to provide standard built-in solutions to problems we are still learning how to solve. In the mean time, there is no reason we shouldn't use Ada to build custom solutions to these problems, just as we would assembly language or any other language. The key issue here is that we must provide an adequate set of primitives, and the freedom to use these primitives to design new solutions, without imposing arbitrary restrictions. This is the kind of change we should hope to see in Ada9x, but we cannot wait.

3.3 Implementation-Defined Extensions

In the short term, we can only take advantage of the liberties given to implementations by the existing Ada standard. These include the right to define new attributes, pragmas, and packages. This freedom is sufficient to allow the extensions we need to work around every shortcoming of the standard language. However, the implementation of some of these extensions will need to be integrated at a low level with the rest of the Ada language implementation.

If there is sufficient demand, the Ada compiler vendor may provide the needed extensions as part of a standard product. This is increasingly likely as compiler vendors begin to write their own runtime systems in Ada, since runtime systems have many of the problematic characteristics of real-time systems. Other factors influencing vendors to provide needed extensions include special requests from customers, and market pressure created by customer awareness. Publication of "standard" extensions like the ARTEWG CIFO [CIFO] can contribute to this market pressure.

As time goes on, one would expect the collection of extensions to reach a point where it is adequate to most users' needs. However, if a vendor does not provide a feature that is needed, the user has the recourse of customization. This can be performed by the compiler vendor, if the vendor has the manpower available and the customer can afford it. Otherwise, the application builder may need to perform this customization.

Of course, by introducing implementation-dependent extensions we have lost some of the benefit of a standard language, but we really had no choice. There are well-known ways to lessen this blow. We can design so as to isolate implementation dependencies in a few modules. We should certainly try to share solutions, so as to avoid needless divergence, and for those areas where the need for extensions is well accepted we can establish ancillary standards.

3.4 Customization for Specific Hardware

There is another good and independent reason for wanting to customize an Ada language implementation. This has to do with application-specific hardware dependencies, and changes to these dependencies during the life of the application system. It is a characteristic of real-time systems that they typically include custom or special-purpose hardware, or at least that they connect a variety of hardware components in a configuration specific to the application. The hardware configuration may change between versions of one application, several versions of which may need to be supported simultaneously. Any off-the-shelf Ada compilation system could hardly be expected to meet all such needs. Some customization will be necessary.

3.5 Tailoring and Configuration

There are several approaches to customization. One, which we have already mentioned, is to pay the compiler vendor to do it. In that case, the user need not be concerned with how it is done. The chief drawback to this approach is the cost, but there are many other reasons the user may prefer to do his own customization. Some examples of such reasons include concern that manpower problems might prevent the vendor from meeting user deadlines, continually evolving requirements due to a prototyping mode of development, or a desire to keep details of the application hardware architecture secret.

Few application developers are prepared to tackle the customization of a complete Ada compilation system. (For those who are, there may be nothing further to discuss.) On the other hand, most application developers have some expertise in building or customizing executives and smaller software tools, such as linkers and debuggers. Thus, some of them are chosing to to buy source licenses from compiler vendors and tackle customization of small components of the compilation system directly, especially the RTS.

Let us call the actual modification of the code of the Ada compilation system "tailoring", because it is analogous to the kind of alteration a tailor does to fit a ready-made suit to a customer, which involves cutting and stitching. Let us contrast this with a less extreme form of adjustment, which we call "configuration" in which the user chooses options and supplies parameters within a scheme set up by the compilation system. (This is analogous choosing the best-fitting jacket and and the best-fitting pants from two racks containing different sizes and styles, or using a belt to adjust the fit of clothing that is too loose, so that no alteration is necessary.)

Clearly, the division of a compilation system into modules with parameters and clearly defined interfaces can assist in both tailoring and configuration. If tailoring is required, clean module interfaces can help to localize changes to a single module, and if an adequate choice of parameters or alternate module versions are provided customization may reduce to choosing among these options.

Most compiler vendors provide some combination options and parameters for configuring their compilers and runtime systems, if only for their own convenience in supporting multiple host and target combinations. This is a good starting point, and as time goes on we can expect vendors to provide more such configuration options, given sufficient pressure from customers. It is inconceivable, however, that any set of configuration choices will be adequate to handle the needs of all applications. Some tailoring (i.e. custom coding) will be necessary.

The RTS is the part of an Ada compilation system most likely to need tailoring. This is good news, because it is much smaller and simpler than an Ada compiler, and can be designed so as to be separated very cleanly from the rest of the compilation system. How difficult it is to tailor the RTS depends on the modularity of the RTS design, the quality of the documentation available, and how often the compiler vendor changes key interface conventions. It can be a large and difficult job, or very simple.

3.6 Alternate Models of Concurrency

One of the main reasons for tailoring the Ada RTS is to support an extended or entirely alternate model of concurrency. As we have mentioned, there are good reasons for dissatisfaction with the minimal standard Ada tasking model.

Some real-time system developers are choosing to provide their own executives, which execute multiple Ada procedures concurrently. This is permitted by the Ada standard on the basis that these procedures are viewed as separate main programs. This capability can be provided alongside Ada tasking. The Lace executive [Lace] represents one way of doing this. Lace offers, more or less, a stripped-down implementation of Ada tasks, so that no execution price is paid for unused functionality. If the implementation of full Ada tasking is constructed on top of Lace, the user has the option of executing any combination of Ada procedures as Lace threads concurrently with normal Ada tasks. The Lace dispatcher controls the allocation of the processor time to both Lace threads and Ada tasks.

The IEEE POSIX portable operating system interface and and the IEEE MOSI microcomputer system interface standards are examples of other program-based models of concurrency competing with Ada tasking. Another is the Distributed Ada Real-time Kernel (DARK), currently under development in the Software Engineering Institute at Carnegie-Mellon University.

Examples of extended task-based models of concurrency are the implementation of alternate rendezvous and scheduling models based on priority inheritance at the Software Engineering Institute and (independently) by DDC-I compiler vendors. Other tasking extensions that have been reportedly implemented via RTS tailoring include non-waiting entry calls, task restarts, and distributed execution. This list will grow.

3.7 Standard Interfaces

A key element in developing extended or alternate Ada runtime systems (RTS's) depends on well-defined interfaces. There are two main reasons for having such interfaces. The first is to provide a means for the user (i.e., application programmer) to communicate with the extended RTS. The second reason is to permit modifications and extensions to the RTS without concern for the internals of the Ada compiler.

Levels of Interface

In separating these concerns, it is useful to distinguish the different levels at which such interfaces are needed:

1. *application-RTS* – the interface which allows a programmer to invoke RTS services explicitly. Such a request would ordinarily be for a service not provided by standard Ada, such as to raise an exception in some Ada task, or to execute some task periodically. This is the level of interface at which the user can take control directly.

2. *compiler-RTS* – the interface by which code generated by the compiler implicitly requests RTS services, in order to implement primitive Ada constructs. Such requests would include delaying the execution of a task, creating a new task, and calling an entry of a task. These services are standard, but may involve implementation choices over which a user may desire extra control.

   The only constraints imposed on the RTS implementor by the Ada language are at this level.

3. *RTS-RTS* – the interface between RTS modules, by which different components of the RTS communicate and request service from one another. Examples of such interfaces include the mechanisms by which the tasking RTS can cause an exception to be raised in a task, and by which the delay implementation obtains information about the passage of time.

Constraints which the RTS imposes on itself at this level can be helpful in localizing application-specific modifications to the RTS. This level of interface can be especially useful to the extent that it separates functions of the RTS that are completely determined by the Ada language from those whose functions are only partly determined, and from those that are not directly connected with the language implementation.

4 The ARTEWG MRTSI

The Model Runtime System Interface (MRTSI) is a document describing a model compiler-RTS interface, and some elements of an RTS-RTS interface. This interace isolates the RTS components directly concerned with the implementation of tasking from compiler and other key RTS components. It's intended purpose is to "serve as a model of clean delineation and explicit documentation of the interface to an Ada runtime system", in hopes that "this can contribute to the production of Ada runtime systems that are better suited to the needs of diverse applications, especially real-time embedded applications, ... provide a pedagogical service to both application builders and compiler vendors by providing a baseline and common frame of reference, in a manner similar to that in which the OSI model serves the communication community", and "encourage greater commonality in RTS interfaces, without going so far as to impose a formal standard." [MRTSI]

4.1 Status of the MRTSI

The MRTSI is still evolving. It has gone through several revisions, starting from an original draft produced in the summer of 1987, and Version 2.3 has been submitted for publication in the ACM SIGAda newsletter *Ada Letters. All specific references to features of the MRTSI in this report refer to the current version at the time this report was written, MRTSI Version 1.5.1.*

Input to the MRTSI has been provided by twenty-five individuals, with experience writing Ada compilers, Ada runtime systems, operating systems, and real-time application programs. Some of this input wa.. provided directly, at meetings, and some of it was provided in the form of written critical reviews, received in response to a broadcast mailing of Version 1.4 to known Ada compiler developers. The author of this report served as collator of these reviewers' comments, as well as leader of the MRTSI Task Force within the ARTEWG, and principal editor of the MRTSI document.

Ideally an RTS interface specification like the MRTSI would be sufficiently precise to permit interoperability of compilers and RTS implementations; that is, any RTS implementation that adheres to the interface would work correctly with any compiler that also conforms to the interface. The present MRTSI draft is too loosely specified to achieve this. If it were tightened up enough to achieve interoperability, it would probably need a specific variant for each target processor and memory architecture.

Another obstacle to a tighter interface specification is the compiler vendors. Most compiler vendors seem opposed to the idea of a standard RTS interface in any form, and their opposition to even a "model" interface seems to grow in proportion to its restrictiveness. Their first main objection is the cost of converting their present (dissimilar) RTS interfaces so that they conform. Another objection is that they hope to achieve some market advantage through proprietary RTS features, which they neither wish to disclose for inclusion in a standard interface, nor be obliged to do without. Their third main objection is that they simply do not want to limit their future options based on today's understanding of runtime systems.

4.2 Prototyping the MRTSI

In order to increase our level of confidence in the MRTSI, that it can indeed be implemented efficiently and does support Ada tasking correctly, we constructed and tested prototype implementations of Version 1.5.1. This made heavy use of code from an existing implementation of Corset[Corset], an Ada runtime system interface designed by T. P. Baker with support of the Boeing Aerospace Company and the Boeing Commercial Airplane Company. (Development of the original prototype Corset implementation was done in 1987 at the Florida State University under contract tọ the Boeing Commercial Airplane Company under Purchase Order Y-429341-0957N)[1].

4.3 Differences Between Corset and the MRTSI

Because Corset and the MRTSI are functionally similar (like any two Ada RTS implementations), it was not very difficult to modify the Corset implementation to conform to the MRTSI interface (Version 1.5.1). Of course, there were quite a few superficial changes, such as in names, data type declarations, and parameters, but the only changes that required modification of the functional code and internal data structures of the prototype RTS were:

1. *Rendezvous.* In Corset, the bodies of **accept** statements are compiled as procedures. The RTS calls one of these procedures when it is time for a rendezvous to take place. Completion of the rendezvous is implicit, through return to the RTS. In the MRTSI, **accept**-bodies are compiled in-line. The RTS begins a rendezvous by returning control to the compiled code, which executes the **accept**-body and then calls the RTS again at the end of the rendezvous. This requires two new RTS service calls: one for normal end of rendezvous, and the other for end of rendezvous due to an unhandled exception (in which the exception is propagated to the calling task).

2. *Task elaboration checks.* In Corset, the compiler-generated code is entirely responsible for checking that when a task is activated the corresponding task body declaration has been elaborated previously. In the MRTSI, this responsibility is shared by the RTS.[2]

3. *Activation sets.* In Corset, the compiler-generated code is entirely responsible for keeping track of sets of tasks that are to be activated together. A set of tasks that are to be activated

[1] The Ada code of the modified Prototype Corset Implementation Software, as reported here, is copyrighted by T.P. Baker and the Florida State University, and is released subject to specific provisions included in the distribution tape. A copy of a tape containing this this Software was delivered to the U.S. Army CECOM with this report.

[2] Contrary to this author's preference for cleaner separation of functions.

together is passed to the RTS via a parameter of an unconstrained array type. By contrast, the MRTSI requires the RTS and the compiler-generated code to cooperate more closely.[2] A set of tasks that are to be activated together is identified by the ID of the last-created member of the set. When a task is created, the compiler-generated code specifies the activation-set to which it should be appended, via the ID of the last-created member of that set. When a set of tasks is to be activated, the set is specified likewise.

4. *Special cases.* The MRTSI goes beyond Corset in distinguishing special cases of "trivial" rendezvous. Implementing these required new code.

4.4 A Problem with Visibility

We discovered that the way in which the MRTSI is divided into several packages limits visibility of key declarations in a way that makes implementation difficult. The problem is that the bodies of the RTS implementation packages need to share access to information that the MRTSI tries to hide from the compiler. This information is in declarations belonging to the bodies and the private parts of the several MRTSI packages. For example, the type TASK_ID exported by package RTS_TASK_IDS is private, and should be so from the viewpoint of compiler-generated code, but the RTS implementation packages need a way to associate attributes with task IDs. While this could theoretically be done using table lookup, that would exact an unacceptable cost in execution time. In our prototype, we therefore made the full declaration of type TASK_ID visible.

A more difficult problem is how to initialize data structures that belong to package RTS_RENDEZVOUS when a new task is created by procedure CREATE_TASK, in package RTS_STAGES. We know of only two ways to solve this kind of problem. One is to add a new operation, that is not part of the MRTSI, to package RTS_RENDEZVOUS. The other is to put such extra operations in a hidden "twin" package, say RTS_RENDEZVOUS_IMPLEMENTATION, which would also contain duplicates of all the declarations in RTS_RENDEZVOUS. The operations exported by RTS_RENDEZVOUS would then be implemented by operations of RTS_RENDEZVOUS_IMPLEMENTATION, using UNCHECKED_CONVERSION to circumvent any type incompatibilities introduced by the duplicate declarations.

Creating twins for the MRTSI packages, as in the example above, can preserve the purity of the MRTSI package specifications at the expense of very ugly code in the package bodies and their twins. Moreover, at the present state of Ada compiler development, this solution is almost certain to result in unacceptable execution-time overhead, due to extra layers of procedure calls. We therefore chose to add what declarations we needed to the MRTSI packages, and to convert some private type declarations into RTS-implementation dependent full type declarations. (See Appendix A for the expanded package specifications as used in the prototype.)

4.5 Testing the MRTSI Prototype

Converting the test programs developed for Corset to conform to the MRTSI was more time-consuming than converting the RTS implementation. Because there is no Ada compiler that complies with the Corset (or the MRTSI) interface, the test programs were originally hand-translated from programs written in full Ada (including tasking). This hand translation had to be redone to conform to the MRTSI.

Because we wanted this prototype to be portable, the test programs are translated into a simple subset of Ada (without tasking). They make explicit calls to the RTS to perform tasking operations where an Ada compiler would insert such calls implicitly when translating Ada tasking constructs. Further, because we needed to be able to simulate concurrency (both between tasks and between multiple processors) and other effects that normally would be achieved by machine-dependent and compiler-dependent code, the translation is rather complex. All procedures, task bodies, and accept statement bodies are converted to blocks of code in a single massive case statement; all variables and parameters are converted to components of a massive variant record type; lexical nesting effects are simulated via explicit chaining of these activation-records.

An example of the translation is sketched below. Two complete examples are given in Appendix B.

```
package PKG is
  task T is entry E; end T;
end PKG;

with TEXT_IO; use TEXT_IO;
package body PKG is
  task body T is
  begin accept E do PUT_LINE("in rendezvous on E"); end E;
  end T;
end PKG;

with PKG; use PKG;
procedure test is begin T.E; end test;
```

This is translated into following code for the main program and body of task T:

```
  ...
  MAIN:        constant PROCEDRUE:= 2;
  T_PROC:      constant PROCEDRUE:= 5;
  ACC_E_PROC:  constant PROCEDRUE:= 6;
  ...
  begin case P is
        when MAIN=>
           --  task T;
           RTS_STAGES.CREATE_TASK(
              SIZE=> RTS_STAGES.SIZE_TYPE(DA'size/STORAGE.UNIT),
              PRIO=> 1,
              NUM_ENTRIES=> 1,
              MASTER=> RTS_STAGES.CURRENT_MASTER,
              STATE=> (T_PROC,COERCE(DA),NULL_DATA_AREA),
              LAST_CREATED=> NULL_TASK,
              ELABORATED=> ELABORATED,
              CREATED_TASK=> DA.T);
           RTS_STAGES.COMPLETE_ACTIVATION;
           RTS_STAGES.ACTIVATE_TASKS(DA.T);
           -- T.E;
           RTS_RENDEZVOUS.CALL_SIMPLE(ACCEPTOR=>DA.T,
                                      E=> 1,
                                      PARAMETER=>(NULL_DATA_AREA,0));
           RTS_STAGES.COMPLETE_TASK;
        when T_PROC=>
           RTS_STAGES.COMPLETE_ACTIVATION;
```

```
     -- accept E do
     RTS_RENDEZVOUS.ACCEPT_CALL(E=> 1,
                                PARAMETER=> DA.PARAMETER);
     begin CHECK(2);
           RTS_RENDEZVOUS.COMPLETE_RENDEZVOUS;
     exception
      when others=> RTS_RENDEZVOUS.EXCEPTIONAL_COMPLETE_RENDEZVOUS
                     (RTS_EXCEPTIONS.CURRENT_EXCEPTION);
     end;
     RTS_STAGES.COMPLETE_TASK;
  when ACC_E_PROC=>
     PUT_LINE("in rendezvous on E");
     ...
end case;
...
```

Each executable program unit corresponds to one alternative of the case statement, which is indexed by a parameter P of type PROCEDRUE. The variable DA points to a local data area which has been allocated for the program unit that is executing. All the standard Ada task structures have been replaced by explicit MRTSI calls, such as would be generated in a translation of the program by a compiler that conforms to the MRTSI interface. The executions of the two programs should therefore be the same, up to allowed implementation-dependencies such as the task dispatching policy.

Some effort was made to produce a program to perform this translation of test programs. Such a translation tool would have allowed us to generate test programs quickly for various RTS interfaces from any test program written in standard Ada. Specifically, we investigated the feasibility of modifying the FSU/AFATL Ada compiler for this purpose. This compiler was designed to translate from Ada to Z8002 assembly code. For a moderate investment of effort (as compared with writing such a translator from scratch), it should be possible to convert this compiler to output simple (non-tasking) Ada code instead of Z8002 assembly code, with the necessary calls to the MRTSI for tasking operations. Nevertheless, it looked as if two man-months or more would be required to complete such a program, so we were forced to be satisfied with limited hand-translation.

Because converting test programs to the MRTSI interface by hand was so labor-intensive, only a few were converted. However, those that were converted all worked correctly without detecting any bugs in the MRTSI implementation, beyond typos of the most trivial kind. This was expected, since the differences between the Corset and MRTSI prototypes were superficial; i.e., limited to the interface. After all, any two Ada RTS implementations should be semantically equivalent, so far as can be detected by a correct Ada program.

4.6 Lessons Learned from Testing

The information gained about the MRTSI was mostly positive. It apparently works as intended. Some simple syntax problems were discovered in the package specifications, but these were easily corrected. Conversion of our RTS implementation to the MRTSI was not difficult.

On the negative side, we discovered two things of a subjective nature about the MRTSI, which will be reported to the ARTEWG MRTSI Task Force:

1. Performing elaboration checks for task bodies adds considerable overhead to task creation and activation, that must be paid even in cases where it can be verified at compile that no elaboration check is needed. This is sad because cases were runtime elaboration checks for tasks are needed appear to be unusual.
2. Supporting an efficient implementation of the "trivial" cases of rendezvous appeared to require adding some overhead to the normal cases.

5 Conclusions

After some early developmental difficulties, Ada seems to be maturing into a language that can be used for real-time software. However, the minimum features required by the Ada standard are insufficient. Producing effective real-time systems requires extended features that may be legally provided by an Ada language implementation within the constraints of the Ada standard.

Since solutions to most of the shortcomings presently perceived in Ada as a real-time language can be solved within the latitude permitted language implementations, it is convenient to view these as shortcomings of implementations rather than of the language. We thus sidestep the organizational difficulties and delays required for language changes, as well as the more serious difficulties of anticipating the results of ongoing research in real-time systems and the special needs of future real-time applications.

In the longer run, it appears desirable to work toward standardization of extensions which have been implemented, tested, and found to meet requirements of a large class of applications, and which are not yet satisfied by other standard mechanisms. In a few cases the standardization vehicle might be the Ada language, but in others cases separate ancillary standards may be better. Such an ancillary standard could define a collection of runtime-system packages with logically related functions, analogous to the CAIS[CAIS].

We hope that a good place to begin to define such ancillary standards is with a compiler-RTS interface specification, and in particular the MRTSI. If such a standard were even adopted by even a few compiler vendors it could give more impetus to the development of specialized Ada runtime environments.

The MRTSI appears to be converging to a form that will be acceptable to both the ARTEWG members and some Ada compiler developers. Based on the experiments we performed, it seems readily implementable. It may be sufficiently close to some existing Ada implementations that they could be converted to adhere to it rather easily. However, most compiler vendors seem to oppose any form of compiler-RTS interface standard.

The present MRTSI draft is too loosely specified to support interoperability of compilers and RTS implementations. If it were tightened up enough to achieve interoperability, it would probably need a specific variant for each target processor and memory architecture. This may eventually be possible. In the mean time, the MRTSI will be useful as a reference model, guideline for interface separation, and basis for discussion of RTS's between compiler producers and users requiring tailoring.

There may be more hope for standardization of other RTS interfaces. One example would be a standard application-RTS interface for low-level tasking operations. If such a package were available it might be used by application programmers to implement their own scheduling policies. This kind of interface might be more directly useful to real-time application programmers than is the MRTSI. A standard interface would also be practical at the RTS-RTS level, between certain key RTS functions which require user tailoring and the rest of the RTS. RTS functions that might be isolated in this way include task scheduling, dynamic storage allocation, and real-time clock services, and interrupt handling.

References

[MRTSI] ARTEWG, "A Model Run Time System Interface", draft reports (Version 1.5.1), ACM SIGAda ARTEWG (1988).

[CIFO] ARTEWG, Ada Run Time Environment Working Group(ACM SIGAda), "A Catalog of Interface Features and Options for the Ada Run Time Environment", Release 2.0, ACM SIGAda (1988).

[Interf] T.P. Baker, "An Improved Ada Runtime System Interface", TR 86-07-05, Computer Science Department, University of Washington, Seattle(July 1986).

[Corset] T.P. Baker, "A Corset for Ada", technical report TR-86-09-06, Computer Science Department, University of Washington, Seattle, WA (1986).

[Lace] T.P. Baker and K. Jeffay, "A Lace for Ada's Corset", technical report TR-86-09-05, Computer Science Department, University of Washington, Seattle, WA (1986).

[Lace2] T.P. Baker, "A Low-level Tasking Package for Ada", *Using Ada: ACM SIGAda International Conference*, Boston, MA (December 1987) 141-146.

[Rat] J.D. Ichbiah, et al, "Rationale for the Design of the Ada Programming Language", SIGPLAN Notices 14,6B ACM (June 1979).

[LSS] J.P. Lehoczky, L. Sha, and J.K. Strosnider, "Aperiodic Scheduling in A Hard Real-Time Environment", Computer Science technical report, Carnegie-Mellon University, Pittsburgh (1987).

[LNL] K.J. Lin, S. Natarajan, and J.W.S. Liu, "Imprecise Results: Utilizing Partial Computations in Real-Time Systems", Proceedings of the 1987 IEEE Real-Time Systems Symposium, IEEE (December, 1987) 210,217.

[CSLRT] D. Cornhill, et al, "Limitations of Ada for Real-Time Scheduling", *International Workshop on Real-Time Ada Issues, Moretonhampstead, Devon, UK*: *Ada Letters* VII,6 ACM (Fall 1987) 33-39.

[Ada83] U.S. Department of Defense, *Military Standard Ada Programming Language*, ANSI/MIL-STD-1815A, Ada Joint Program Office (January 1983).

[CAIS] U.S. Department of Defense, *Military Standard Common APSE Interface Set (CAIS)*, Proposed MIL-STD-CAIS, Ada Joint Program Office (January 1985).

[Iron] U.S. Department of Defense, "Department of Defense Requirements for High Order Computer Programming Languages", *SIGPLAN Notices*, 12,12 (December 1977).

A MRTSI Prototype Specifications

The following are the package specifications of the MRTSI, as modified to include declarations needed by the bodies (i.e., implementations) of the other MRTSI packages[3]

```
package RTS_TASK_IDS is
  type TASK_ID is                -- defined by RTS implementation
       range 0..31;              -- for prototype RTS implementation
  NULL_TASK: constant TASK_ID:=  -- defined by RTS implementation
       0;                        -- for prototype RTS implementation
  function SAME_TASK(L,R: TASK_ID) return BOOLEAN renames "=";
  ---------end of MRTSI view--------------------
  subtype NONNULL_TASK_ID is TASK_ID range 1..31;
end RTS_TASK_IDS;

package RTS_EXCEPTIONS is
  type EXCEPTION_ID is private;
  type EVENT_ID is private;
  NULL_EXCEPTION_ID: constant EXCEPTION_ID;
  CONSTRAINT_ERROR_ID: constant EXCEPTION_ID;
  NUMERIC_ERROR_ID: constant EXCEPTION_ID;
  PROGRAM_ERROR_ID: constant EXCEPTION_ID;
  STORAGE_ERROR_ID: constant EXCEPTION_ID;
  TASKING_ERROR_ID: constant EXCEPTION_ID;
  procedure RAISE_EXCEPTION(E: EXCEPTION_ID);
  function  CURRENT_EXCEPTION return EXCEPTION_ID;
  procedure NOTIFY_EXCEPTION(EVENT: EVENT_ID);
---------------end of MRTSI view-----------------------------
private --depends on compiler and machine.
  type EXCEPTION_ID is new INTEGER;
  type EVENT_ID is range 0..31;
  NULL_EXCEPTION_ID: constant EXCEPTION_ID:= 0;
  CONSTRAINT_ERROR_ID: constant EXCEPTION_ID:= 1;
  NUMERIC_ERROR_ID: constant EXCEPTION_ID:= 2;
  PROGRAM_ERROR_ID: constant EXCEPTION_ID:= 3;
  STORAGE_ERROR_ID: constant EXCEPTION_ID:= 4;
  TASKING_ERROR_ID: constant EXCEPTION_ID:= 5;
end RTS_EXCEPTIONS;

with SYSTEM;
with RTS_EXCEPTIONS; use RTS_EXCEPTIONS;
with RTS_TASK_IDS; use RTS_TASK_IDS;
with RTS_INTERRUPTS; use RTS_INTERRUPTS;
with MACHINE; -- only for implementation view
package RTS_RENDEZVOUS is
  NULL_ENTRY: constant:= 0;
  MAX_ENTRY: constant:=       -- value defined by RTS implementation
                        255; -- for prototype RTS implementation
  type ENTRY_INDEX is range NULL_ENTRY .. MAX_ENTRY;
  type CALLER_PARAMETER_DESCRIPTOR is   -- machine-specific
       record LOC: MACHINE.DATA_AREA;   -- for prototype RTS implementation
              LEN: NATURAL;
       end record;
  type ACCEPTOR_PARAMETER_DESCRIPTOR is -- machine-specific
```

[3] These package specifications are copyrighted by the T.P. Baker and the Florida State University, and now are released to the public domain.

```
     record LOC: MACHINE.DATA_AREA;   -- for prototype RTS implementation
     end record;
-- Lower index bound is 1 by convention.
procedure CALL_SIMPLE(ACCEPTOR: TASK_ID;
                      E: ENTRY_INDEX;
                      PARAMETER: CALLER_PARAMETER_DESCRIPTOR);
procedure CALL_CONDITIONAL(ACCEPTOR: TASK_ID;
                           E: ENTRY_INDEX;
                           PARAMETER: CALLER_PARAMETER_DESCRIPTOR;
                           RENDEZVOUS_SUCCESSFUL: out BOOLEAN);
procedure CALL_TIMED(ACCEPTOR: TASK_ID;
                     E: ENTRY_INDEX;
                     PARAMETER: CALLER_PARAMETER_DESCRIPTOR;
                     D: DURATION;
                     RENDEZVOUS_SUCCESSFUL: out BOOLEAN);
procedure TRIVIAL_CALL(ACCEPTOR: TASK_ID;
                       E: ENTRY_INDEX;
                       RENDEZVOUS_SUCCESSFUL: out BOOLEAN);
type SELECT_INDEX is new INTEGER;
NO_RENDEZVOUS: constant SELECT_INDEX:= 0;
type ENTRY_LIST is array (SELECT_INDEX range <>) of ENTRY_INDEX;
type MODES is (SIMPLE_MODE, DELAY_MODE, ELSE_MODE, TERMINATE_MODE);
procedure SELECTIVE_WAIT(OPEN_ENTRIES: ENTRY_LIST;
                         D: DURATION;
                         SELECT_MODE: MODES;
                         PARAMETER: out ACCEPTOR_PARAMETER_DESCRIPTOR;
                         INDEX: out SELECT_INDEX);
procedure ACCEPT_CALL(E: ENTRY_INDEX;
                      PARAMETER: out ACCEPTOR_PARAMETER_DESCRIPTOR);
procedure TRIVIAL_ACCEPT(E: ENTRY_INDEX);
function  COUNT(T: TASK_ID; E: ENTRY_INDEX) return NATURAL;
function  CALLABLE(T: TASK_ID) return BOOLEAN;
procedure ASSOCIATE_INTERRUPT(INTERRUPT: INTERRUPT_ID;
                              ACCEPTOR: TASK_ID;
                              E: ENTRY_INDEX);
procedure DISSOCIATE_INTERRUPT(INTERRUPT: INTERRUPT_ID);
procedure COMPLETE_RENDEZVOUS;
procedure EXCEPTIONAL_COMPLETE_RENDEZVOUS(E: EXCEPTION_ID);

package INTERRUPT_BUFFERS is
  type BUFFER_ARRAY is array (INTEGER range <>) of INTEGER;
  type BUFFER_ACCESS is access BUFFER_ARRAY;
  type COLLECTION_ID is private;
  function NEW_COLLECTION(ACCEPTOR: TASK_ID;
                          E: ENTRY_INDEX;
                          NUMBER: POSITIVE;
                          SIZE: POSITIVE) return COLLECTION_ID;
  function NEW_BUFFER(COLLECTION: COLLECTION_ID) return BUFFER_ACCESS;
  procedure ENQUEUE(BUFFER: BUFFER_ACCESS);
private
  type COLLECTION_ID is new INTEGER;
end INTERRUPT_BUFFERS;
---------end of MRTSI view--------------------
function  CALLING_TASK return TASK_ID;
procedure OPEN_ENTRIES(T: TASK_ID; NUM_ENTRIES: INTEGER);
procedure CLOSE_ENTRIES(T: TASK_ID);
procedure CALL_OFFLINE(A: TASK_ID;
                       E: ENTRY_INDEX;
                       LOC: MACHINE.DATA_AREA);
```

```
end RTS_RENDEZVOUS;

with RTS_TASK_IDS; use RTS_TASK_IDS;
with MACHINE;  -- for this prototype implementation
package RTS_STAGES is
  type MASTER_ID is private;
  type ACCESS_BOOLEAN is access BOOLEAN;
  function  CURRENT_MASTER return MASTER_ID;
  procedure ENTER_MASTER;
  procedure COMPLETE_MASTER;
  type INIT_STATE is        -- machine-specific
       new MACHINE.STATE;   -- for this prototype RTS implementation
  type SIZE_TYPE is         -- machine-specific
       range 0..1023;       -- for this prototype RTS implementation
  UNSPECIFIED_SIZE: constant SIZE_TYPE:= SIZE_TYPE'first;
  procedure CREATE_TASK(SIZE: SIZE_TYPE;
                        PRIO: INTEGER;
                        NUM_ENTRIES: NATURAL;
                        MASTER: MASTER_ID;
                        STATE: INIT_STATE;
                        LAST_CREATED: TASK_ID;
                        ELABORATED: ACCESS_BOOLEAN;
                        CREATED_TASK: out TASK_ID);
  procedure ACTIVATE_TASKS(LAST_CREATED: TASK_ID);
  procedure COMPLETE_ACTIVATION;
  procedure COMPLETE_TASK;
  function  TERMINATED(T: TASK_ID) return BOOLEAN;
  function  IS_LOCAL_TASK(T: TASK_ID;
                          MASTER: MASTER_ID) return BOOLEAN;
  ---------end of MRTSI view--------------------
  function ANCESTOR(T: TASK_ID; LEVELS_OUT: INTEGER) return TASK_ID;
  -- used in abortion.
  procedure COMPLETE(T: TASK_ID);
  -- used in abortion.
  procedure TERMINATE_ALTERNATIVE;
  -- used in rendezvous.
private
  type MASTER_ID is -- defined by the RTS implementation
                 new INTEGER;
end RTS_STAGES;

package RTS_CLOCK is
  type DAYS is range 0.. (2**31)-1;
  subtype DAY_DURATION is DURATION range 0.0 .. 86_400.0;
  -- These ranges are only examples.
  type TIME is
    record DAY: DAYS;
           SECOND: DAY_DURATION;
    end record;
  -- TICK: constant:= 2.0**(-14);
  -- This value of TICK is only an example.
  function CLOCK return TIME;
  ---------end of MRTSI view--------------------
  --TICKS_PER_SECOND: constant:= 2**14;
  TICKS_PER_SECOND: constant:= 2;
  --TICK set large, for prototype RTS implementation.
  TICK: constant:= 1.0/TICKS_PER_SECOND;
  type TICKS is private;
  function CLOCK return TICKS;
  function CLOCK_IS_PAST(T: TICKS) return BOOLEAN;
  function "+"(L: TICKS; R: INTEGER) return TICKS;
  procedure STOP; -- shuts down simulation.
  procedure INTERRUPT; -- simulates clock tick.
private
  type TICKS is range -(2**31)..(2**31-1);
end RTS_CLOCK;

with RTS_CLOCK; use RTS_CLOCK;
with RTS_TASK_IDS; use RTS_TASK_IDS; -- only for implementation view
package RTS_DELAYS is
  procedure DELAY_SELF(D: DURATION);
  ---------end of MRTSI view--------------------
  procedure INTERNAL_DELAY_SELF(D: DURATION);
  TIME_OF_NEXT_CHECK: RTS_CLOCK.TICKS;
  procedure CHECK;
  procedure CLAIM_RIGHT_TO_CANCEL_DELAY(T: TASK_ID; TAKEN: out BOOLEAN);
  procedure OPEN_DELAYS(T: TASK_ID);
  procedure CLOSE_DELAYS(T: TASK_ID);
end RTS_DELAYS;

with SYSTEM;
with MACHINE; -- for implementation view only
package RTS_STORAGE_MANAGEMENT is
  type COLLECTION_ID is private;
  function NEW_COLLECTION(COLLECTION_SIZE: NATURAL:=0;
                          MAX_BLOCK_SIZE: NATURAL:=0)
                          return COLLECTION_ID;
--  function NEW_BLOCK(SIZE: NATURAL;
--                     COLLECTION: COLLECTION_ID)
--                     return SYSTEM.ADDRESS;
--  procedure RELEASE_BLOCK(LOCATION: SYSTEM.ADDRESS);
  function NEW_BLOCK(SIZE: NATURAL;
                     COLLECTION: COLLECTION_ID) return MACHINE.DATA_AREA;
  procedure RELEASE_BLOCK(LOCATION: MACHINE.DATA_AREA);
  procedure RELEASE_COLLECTION(COLLECTION: COLLECTION_ID);
  ---------end of MRTSI view--------------------
  NULL_COLLECTION: constant COLLECTION_ID;
private
  type COLLECTION_ID is range 0..(2**15)-1;
  NULL_COLLECTION: constant COLLECTION_ID:= 0;
end RTS_STORAGE_MANAGEMENT;

with RTS_TASK_IDS; use RTS_TASK_IDS;
with MACHINE; -- for prototype RTS implementation
package RTS_ABORTION is
  type TASK_LIST is array (POSITIVE range <>) of TASK_ID;
  procedure ABORT_TASKS(TASKS: TASK_LIST);
  ---------end of MRTSI view--------------------
  procedure CLAIM_RIGHT_TO_RELEASE(T: TASK_ID;
                                   TAKEN: out BOOLEAN);
  procedure FORESTALL_ABORTION;
  procedure SAFE_RELEASE(T: TASK_ID);
  procedure SAFE_HOLD;
  procedure SAFE_FORCE_CALL(T: TASK_ID;
                            P: MACHINE.PROCEDRUE);
  procedure OPEN_ABORTION(T: TASK_ID);
end RTS_ABORTION;
```

```
with SYSTEM;
with RTS_TASK_IDS; use RTS_TASK_IDS;
package RTS_INTERRUPTS is
  type INTERRUPT_ID is -- machine-specific
       range 0..255;    -- for this prototype RTS implementation
  procedure BIND_HANDLER(INTERRUPT:INTERRUPT_ID; HANDLER: SYSTEM.ADDRESS);
  procedure UNBIND_HANDLER(INTERRUPT:INTERRUPT_ID);
  function  NULL_HANDLER return SYSTEM.ADDRESS;
  procedure SOFTWARE_INTERRUPT(INTERRUPT: INTERRUPT_ID);
  procedure ATTEMPT_PREEMPTION;
end RTS_INTERRUPTS;

package RTE_PRIORITIES is
  MIN_PRIORITY: constant:= 1;
  MAX_PRIORITY: constant:= 31;
end RTE_PRIORITIES;

with RTS_INTERRUPTS; use RTS_INTERRUPTS;
with RTS_RENDEZVOUS; use RTS_RENDEZVOUS;
with RTS_TASK_IDS; use RTS_TASK_IDS;
package QUEUED_INTERRUPTS is
  procedure ASSOCIATE_INTERRUPT(INTERRUPT: INTERRUPT_ID;
                                ACCEPTOR: TASK_ID;
                                E: ENTRY_INDEX);
  procedure DISSOCIATE_INTERRUPT(INTERRUPT: INTERRUPT_ID);
end QUEUED_INTERRUPTS;
```

B Example Test Programs

The following are two versions each of test programs. Each program is shown before and after translation to the form required by the MRTSI prototype. Calls to procedures CHECK() and COMPLETE, from package CHECKS, are to check that the various operations are performed in the correct order.

```
-- test_1: tests creation and normal termination of a collection of
-- dependent tasks.

with SYSTEM;
package PKG_1 is
  task T1 is
    pragma PRIORITY(1);
  end T1;
  task type TYPE2 is
    pragma PRIORITY(2);
  end TYPE2;
  procedure P;
end PKG_1;

with CHECKS; use CHECKS;
with TEXT_IO;
with TRACE_1;
pragma ELABORATE(TRACE_1);
package body PKG_1 is
  X: INTEGER:= 1;
  task body T1 is
```

```
 begin CHECK(1);
       P;
       CHECK(2);
       COMPLETE;
 end T1;
 task body TYPE2 is
 begin CHECK(3);
       if X<=1 then X:=X+1; P; end if;
       CHECK(4);
       COMPLETE;
 end TYPE2;
 procedure P is
   T2: TYPE2;
   type accT is access TYPE2;
 begin CHECK(5);
 B:    declare
         task T3 is
           pragma PRIORITY(3);
         end T3;
         task body T3 is
           procedure R is
             X: accT;
             procedure S is
               T4: TYPE2;
             begin CHECK(6);
                   X:=new TYPE2;
                   CHECK(7);
             end S;
           begin CHECK(8);
                 S;
                 CHECK(9);
           end R;
          begin CHECK(10);
                R;
                CHECK(11);
                COMPLETE;
          end T3;
        begin CHECK(12);
        end B;
  CHECK(13);
  end P;
end PKG_1;

with SYSTEM;
with CHECKS; use CHECKS;
with PKG_1; use PKG_1;
procedure test_1 is
  pragma PRIORITY(0);
begin CHECK(14); COMPLETE;
end test_1;
```

```
-- test_1:  tests normal task termination.
-- This was also useful in checking treatment of recovery from running out
-- of threads, before range of threads was extended.

with RTS_ABORTION;
with LACE;
with RIGHTS;
with RTS_STAGES;
with STORAGE;
with RTS_TASK_IDS;
with UNCHECKED_DEALLOCATION;
with UNCHECKED_CONVERSION;
with TRACE_1;
pragma ELABORATE(TRACE_1);
separate(MACHINE)
package body PROCEDRUES is

  use LACE;
  use MACHINE;
  use RTS_TASK_IDS;

  subtype RIGHT is RIGHTS.RIGHT;
  procedure CLAIM(R: RIGHT; B: out BOOLEAN) renames RIGHTS.CLAIM;
  procedure INIT(R: in out RIGHT;
                 AVAILABLE: BOOLEAN:= TRUE) renames RIGHTS.INIT;

  RIGHT_TO_START_UP: RIGHT;

  -- Procedures (simulated).

  --NULL_PROCEDRUE: constant PROCEDRUE:= 0;
  --STARTUP:   constant PROCEDRUE:= 1;

  MAIN:        constant PROCEDRUE:= 2;
  --TERMINATE_PROC:           constant PROCEDRUE:= 3;
  --TASKING_ERROR_PROC: constant PROCEDRUE:= 4;
  P_PROC:        constant PROCEDRUE:= 5;
  T1_PROC:       constant PROCEDRUE:= 6;
  Type2_PROC:    constant PROCEDRUE:= 7;
  T3_PROC:       constant PROCEDRUE:= 8;
  R_PROC:        constant PROCEDRUE:= 9;
  S_PROC:        constant PROCEDRUE:=10;

  IS_TASK: PROCEDRUE_BIT_VECTOR:=ADA_HAS_SILLY_RESTRICTIONS(
  (MAIN|T1_PROC|Type2_PROC|T3_PROC=> TRUE,
   others=> FALSE));
  LEVEL: PROCEDRUE_INTEGER_VECTOR:=ADA_HAS_SILLY_RESTRICTIONS(
  (NULL_PROCEDRUE=> -1,
   STARTUP=> 0,
   MAIN=> 1,
   TERMINATE_PROC=> 2,
   TASKING_ERROR_PROC=> 2,
   P_PROC=> 2,
   T1_PROC=> 2,
   Type2_PROC=> 2,
   T3_PROC=>3,
   R_PROC=> 4,
   S_PROC=> 5,
   others=> -1));
```

```
type Acc_TASK_ID is access TASK_ID;

-- Procedure activation-record types

type PARAM_RECORD(P: PROCEDRUE);
type PARAM is access PARAM_RECORD;
type STORE_RECORD(P: PROCEDRUE);
type STORE is access STORE_RECORD;
type STORE_RECORD(P: PROCEDRUE) is
     record
          OUTER: STORE;
          PROC:  PROCEDRUE;
          PARAMS: PARAM;
          OLD_DATA: MACHINE.DATA_AREA;
          case P is
     when STARTUP=>
          BUSY: BOOLEAN;
          MAIN_TASK: TASK_ID;
     when MAIN=>
          X: INTEGER:= 1;
          T1: TASK_ID;
     when TERMINATE_PROC=> null;
     when TASKING_ERROR_PROC=>     null;
     when P_PROC=>
          T2: TASK_ID;
          ACCTMASTER: RTS_STAGES.MASTER_ID;
          T3: TASK_ID; -- from block B
     when T1_PROC=>     null;
     when TYPE2_PROC=>     null;
     when T3_PROC=> null;
     when R_PROC=>
          XX: Acc_TASK_ID := new TASK_ID;
     when S_PROC=>
          T4: TASK_ID;
     when others=>    null;
          end case;
     end record;

type PARAM_RECORD(P: PROCEDRUE) is
     record -- add parameter declarations here.
          case P is
     when others=>    null;
            end case;
     end record;

function COERCE is
  new UNCHECKED_CONVERSION(STORE, MACHINE.DATA_AREA);
function COERCE is
  new UNCHECKED_CONVERSION(MACHINE.DATA_AREA, STORE);
function COERCE is
  new UNCHECKED_CONVERSION(PARAM, MACHINE.DATA_AREA);
function COERCE is
  new UNCHECKED_CONVERSION(MACHINE.DATA_AREA, PARAM);
procedure OLD is
  new UNCHECKED_DEALLOCATION(STORE_RECORD,STORE);
procedure OLD is
  new UNCHECKED_DEALLOCATION(PARAM_RECORD,PARAM);
  -- Data for simulated tasks.

  TMP: PARAM;
  ELABORATED: RTS_STAGES.ACCESS_BOOLEAN:= new BOOLEAN'(TRUE);

procedure ALLOCATE(P: PROCEDRUE; A: out MACHINE.DATA_AREA) is
begin A:= COERCE(new STORE_RECORD(P));
end ALLOCATE;

procedure CALL(P: PROCEDRUE; A: MACHINE.DATA_AREA) is
 PA: PARAM renames COERCE(A);
 DA: STORE:= new STORE_RECORD(P);
 ST: STATE renames CURRENT_STATE(HERE.CURRENT_THREAD);
begin DA.OUTER:= COERCE(ST.DATA);
      DA.PROC:= P;
      for I in LEVEL(P)..LEVEL(COERCE(ST.DATA).PROC)
      loop DA.OUTER:=DA.OUTER.OUTER; end loop;
      DA.OLD_DATA:=ST.DATA;
      ST.DATA:= COERCE(DA);
      DA.PARAMS:= PA;
      begin -- frame to insure tasks complete
      case P is
      when STARTUP=>
           NON_ADA(HERE.CURRENT_THREAD):=TRUE;
           CLAIM(RIGHT_TO_START_UP,DA.BUSY);
           if not DA.BUSY
           then RTS_STAGES.CREATE_TASK(
                  SIZE=> RTS_STAGES.SIZE_TYPE(DA'size/STORAGE.UNIT),
                  PRIO=> 0,
                  NUM_ENTRIES=> 0,
                  MASTER=> RTS_STAGES.CURRENT_MASTER,
                  STATE=> (MAIN,COERCE(DA),NULL_DATA_AREA),
                  LAST_CREATED=> NULL_TASK,
                  ELABORATED=> ELABORATED,
                  CREATED_TASK=> DA.MAIN_TASK);
                RELEASE(DA.MAIN_TASK);
           end if;
           loop LACE.IDLE_DISPATCH;
           end loop;
      when MAIN=>
           NON_ADA(HERE.CURRENT_THREAD):=TRUE;
           --  task T1;
           RTS_STAGES.CREATE_TASK(
              SIZE=> RTS_STAGES.SIZE_TYPE(DA'size/STORAGE.UNIT),
              PRIO=> 1,
              NUM_ENTRIES=> 0,
              MASTER=> RTS_STAGES.CURRENT_MASTER,
              STATE=> (T1_PROC,COERCE(DA),NULL_DATA_AREA),
              LAST_CREATED=> NULL_TASK,
              ELABORATED=> ELABORATED,
              CREATED_TASK=> DA.T1);
           -- X: INTEGER:= 1;
           DA.X:= 1;
           RTS_STAGES.COMPLETE_ACTIVATION;
           RTS_STAGES.ACTIVATE_TASKS(DA.T1);
           CHECK(14);
      when P_PROC=>
           RTS_STAGES.ENTER_MASTER;
           -- T2: TYPE2;
```

```
     RTS_STAGES.CREATE_TASK(
        SIZE=> RTS_STAGES.SIZE_TYPE(DA'size/STORAGE.UNIT),
        PRIO=> 2,
        NUM_ENTRIES=> 0,
        MASTER=> RTS_STAGES.CURRENT_MASTER,
        STATE=> (TYPE2_PROC,COERCE(DA),NULL_DATA_AREA),
        LAST_CREATED=> NULL_TASK,
        ELABORATED=> ELABORATED,
        CREATED_TASK=> DA.T2);
     RTS_STAGES.ACTIVATE_TASKS(DA.T2);
     -- type ACCT is access TYPE2;
     DA.ACCTMASTER:=RTS_STAGES.CURRENT_MASTER;
     CHECK(5);
     RTS_STAGES.ENTER_MASTER; -- for block B
     -- task T3;
     RTS_STAGES.CREATE_TASK(
        SIZE=> RTS_STAGES.SIZE_TYPE(DA'size/STORAGE.UNIT),
        PRIO=> 3,
        NUM_ENTRIES=> 0,
        MASTER=> RTS_STAGES.CURRENT_MASTER,
        STATE=> (T3_PROC,COERCE(DA),NULL_DATA_AREA),
        LAST_CREATED=> NULL_TASK,
        ELABORATED=> ELABORATED,
        CREATED_TASK=> DA.T3);
     RTS_STAGES.ACTIVATE_TASKS(DA.T3);
     CHECK(12);
     RTS_STAGES.COMPLETE_MASTER; -- for B
     CHECK(13);
     RTS_STAGES.COMPLETE_MASTER; -- for P
when T1_PROC=>
     RTS_STAGES.COMPLETE_ACTIVATION;
     CHECK(1);
     -- P;
     CALL(P_PROC,NULL_DATA_AREA);
     CHECK(2);
when TYPE2_PROC=>
     RTS_STAGES.COMPLETE_ACTIVATION;
     CHECK(3);
     if DA.OUTER.X<=1
     then DA.OUTER.X:=DA.OUTER.X+1;
          -- P;
          CALL(P_PROC,NULL_DATA_AREA);
     end if;
     CHECK(4);
when T3_PROC=>
     RTS_STAGES.COMPLETE_ACTIVATION;
     CHECK(10);
     -- R;
     CALL(R_PROC,NULL_DATA_AREA);
     CHECK(11);
when R_PROC=>
     CHECK(8);
     -- S;
     CALL(S_PROC,NULL_DATA_AREA);
     -- if there were a parameter:
     --TMP:= new PARAM_RECORD'(..initial values..);
     --CALL(S_PROC,COERCE(TMP));
     --OLD(TMP);
     CHECK(9);
```

```
      when S_PROC=>
          RTS_STAGES.ENTER_MASTER;
          -- T4: TYPE2;
          RTS_STAGES.CREATE_TASK(
             SIZE=> RTS_STAGES.SIZE_TYPE(DA'size/STORAGE.UNIT),
             PRIO=> 2,
             STATE=> (TYPE2_PROC,COERCE(DA),NULL_DATA_AREA),
             NUM_ENTRIES=> 0,
             MASTER=> RTS_STAGES.CURRENT_MASTER,
             LAST_CREATED=> NULL_TASK,
             ELABORATED=> ELABORATED,
             CREATED_TASK=> DA.T4);
          RTS_STAGES.ACTIVATE_TASKS(DA.T4);
          CHECK(6);
          -- X:=new TYPE2;
          RTS_STAGES.CREATE_TASK(
             SIZE=> RTS_STAGES.SIZE_TYPE(DA'size/STORAGE.UNIT),
             PRIO=> 2,
             NUM_ENTRIES=> 0,
             MASTER=> DA.OUTER.OUTER.OUTER.ACCTMASTER,
             STATE=> (TYPE2_PROC,COERCE(DA),NULL_DATA_AREA),
             LAST_CREATED=> NULL_TASK,
             ELABORATED=> ELABORATED,
             CREATED_TASK=> DA.OUTER.XX.all);
          RTS_STAGES.ACTIVATE_TASKS(DA.OUTER.XX.all);
          CHECK(7);
          RTS_STAGES.COMPLETE_MASTER;
      when TERMINATE_PROC=>
          RTS_STAGES.COMPLETE_TASK;
      when TASKING_ERROR_PROC=>
          raise TASKING_ERROR;
      when others=>
          ERROR("undefined PROC. ");
      end case;
      exception when others=>
             ST.DATA:= DA.OLD_DATA;
             OLD(DA);
             raise;
      end;
      if IS_TASK(P) then COMPLETE; RTS_STAGES.COMPLETE_TASK; end if;
      ST.DATA:= DA.OLD_DATA;
      OLD(DA);
end CALL;

procedure DEALLOCATE(A: in out MACHINE.DATA_AREA) is
  AA: STORE:= COERCE(A);
  -- due to restriction RM 6.4.1(3) that actual of OLD must be a variable
  -- name or conversion, where COERCE is not recognized as a conversion.
begin OLD(AA); A:=NULL_DATA_AREA;
end DEALLOCATE;

begin INIT(RIGHT_TO_START_UP,AVAILABLE=> TRUE);
end PROCEDRUES;
```

```
-- test_4: simple rendezvous (no parameters), conditional entry call,
--         and selective wait

with SYSTEM;
package PKG_4 is
  task T1 is
     pragma PRIORITY(1);
     entry E;
     entry STOP;
  end T1;
end PKG_4;

with CHECKS; use CHECKS;
with TEXT_IO;
with TRACE_4;
pragma ELABORATE(TRACE_4);
package body PKG_4 is
  X: INTEGER:= 1;
  task body T1 is
  begin CHECK(1);
        accept E do CHECK(2);
        end E;
        CHECK(3);
        accept E do CHECK(4);
        end E;
        CHECK(5);
        select
            accept E do CHECK(6);
            end E;
            CHECK(7);
        else
            CHECK(8);
        end select;
        select
            accept E do CHECK(9);
            end E;
            CHECK(10);
        else
            CHECK(11);
        end select;
        accept STOP do CHECK(12);
        end STOP;
        COMPLETE;
  end T1;
end PKG_4;

with SYSTEM;
with CHECKS; use CHECKS;
with PKG_4; use PKG_4;
procedure test_4 is
  pragma PRIORITY(0);
begin   CHECK(13);
        T1.E;
        T1.E;
        select
           T1.E; CHECK(14);
        else
           CHECK(15);
        end select;
```

```
          select
             T1.E; CHECK(16);
          else
             CHECK(17);
          end select;
          T1.STOP;
          COMPLETE;
     end test_4;
```

```
-- test_4: tests simple rendezvous (no parameters), conditional entry call,
--         and selective wait

with LACE;
with RTS_ABORTION;
with RIGHTS;
with RTS_TASK_IDS;
with UNCHECKED_DEALLOCATION;
with UNCHECKED_CONVERSION;
with RTS_STAGES;
with RTS_RENDEZVOUS;
with STORAGE;
with RTS_EXCEPTIONS;
with TRACE_4;
pragma ELABORATE(TRACE_4);
separate(MACHINE)
package body PROCEDRUES is

  use LACE;
  use MACHINE;
  use RTS_TASK_IDS;

  subtype RIGHT is RIGHTS.RIGHT;
  procedure CLAIM(R: RIGHT; B: out BOOLEAN) renames RIGHTS.CLAIM;
  procedure INIT(R: in out RIGHT;
                 AVAILABLE: BOOLEAN:= TRUE) renames RIGHTS.INIT;

  RIGHT_TO_START_UP: RIGHT;

  -- Procedures (simulated).

  --NULL_PROCEDRUE: constant PROCEDRUE:= 0;
  --STARTUP:   constant PROCEDRUE:= 1;
  MAIN:        constant PROCEDRUE:= 2;
  --TERMINATE_PROC:         constant PROCEDRUE:= 3;
  --TASKING_ERROR_PROC: constant PROCEDRUE:= 4;
  T1_PROC:       constant PROCEDRUE:= 5;

  IS_TASK: PROCEDRUE_BIT_VECTOR:=ADA_HAS_SILLY_RESTRICTIONS(
  (MAIN|T1_PROC=> TRUE,
   others=> FALSE));
  LEVEL: PROCEDRUE_INTEGER_VECTOR:=ADA_HAS_SILLY_RESTRICTIONS(
  (NULL_PROCEDRUE=> -1,
   STARTUP=> 0,
   MAIN=> 1,
   TERMINATE_PROC=> 2,
   TASKING_ERROR_PROC=> 2,
   T1_PROC=> 2,
   others=> -1));

  type Acc_TASK_ID is access TASK_ID;

  -- Procedure activation-record types

  type PARAM_RECORD(P: PROCEDRUE);
  type PARAM is access PARAM_RECORD;
  type STORE_RECORD(P: PROCEDRUE);
  type STORE is access STORE_RECORD;
  type STORE_RECORD(P: PROCEDRUE) is
```

```
       record
            OUTER: STORE;
            PROC:  PROCEDRUE;
            PARAMS: PARAM;
            OLD_DATA: MACHINE.DATA_AREA;
            case P is
       when STARTUP=>
            BUSY: BOOLEAN;
            MAIN_TASK:TASK_ID;
       when MAIN=>
            T1: TASK_ID;
            RES: BOOLEAN;
       when T1_PROC=>
            PARAMETER: RTS_RENDEZVOUS.ACCEPTOR_PARAMETER_DESCRIPTOR;
            RES2: RTS_RENDEZVOUS.SELECT_INDEX;
       when others=>    null;
            end case;
       end record;

  type PARAM_RECORD(P: PROCEDRUE) is
       record -- add parameter declarations here.
            case P is
       when others=>    null;
              end case;
       end record;

  function COERCE is
    new UNCHECKED_CONVERSION(STORE, MACHINE.DATA_AREA);
  function COERCE is
    new UNCHECKED_CONVERSION(MACHINE.DATA_AREA, STORE);
  function COERCE is
    new UNCHECKED_CONVERSION(PARAM, MACHINE.DATA_AREA);
  function COERCE is
    new UNCHECKED_CONVERSION(MACHINE.DATA_AREA, PARAM);
  procedure OLD is
    new UNCHECKED_DEALLOCATION(STORE_RECORD,STORE);
  procedure OLD is
    new UNCHECKED_DEALLOCATION(PARAM_RECORD,PARAM);


  -- Data for simulated tasks.

  TMP: PARAM;
  ELABORATED: RTS_STAGES.ACCESS_BOOLEAN:= new BOOLEAN'(TRUE);

procedure ALLOCATE(P: PROCEDRUE; A: out MACHINE.DATA_AREA) is
begin A:= COERCE(new STORE_RECORD(P));
end ALLOCATE;

procedure CALL(P: PROCEDRUE; A: MACHINE.DATA_AREA) is
 PA: PARAM renames COERCE(A);
 DA: STORE:= new STORE_RECORD(P);
 ST: STATE renames CURRENT_STATE(HERE.CURRENT_THREAD);
begin DA.OUTER:= COERCE(ST.DATA);
      DA.PROC:= P;
      for I in LEVEL(P)..LEVEL(COERCE(ST.DATA).PROC)
      loop DA.OUTER:=DA.OUTER.OUTER; end loop;
      DA.OLD_DATA:=ST.DATA;
      ST.DATA:= COERCE(DA);
```

```
DA.PARAMS:= PA;
begin -- frame to insure tasks complete
case P is
when STARTUP=>
     NON_ADA(HERE.CURRENT_THREAD):=TRUE;
     CLAIM(RIGHT_TO_START_UP,DA.BUSY);
     if not DA.BUSY
     then RTS_STAGES.CREATE_TASK(
            SIZE=> RTS_STAGES.SIZE_TYPE(DA'size/STORAGE.UNIT),
            PRIO=> 0,
            NUM_ENTRIES=> 0,
            MASTER=> RTS_STAGES.CURRENT_MASTER,
            STATE=> (MAIN,COERCE(DA),NULL_DATA_AREA),
            LAST_CREATED=> NULL_TASK,
            ELABORATED=> ELABORATED,
            CREATED_TASK=> DA.MAIN_TASK);
          RELEASE(DA.MAIN_TASK);
     end if;
     loop IDLE_DISPATCH;
     end loop;

when MAIN=>
     NON_ADA(HERE.CURRENT_THREAD):=TRUE;
     --  task T1;
     RTS_STAGES.CREATE_TASK(
        SIZE=> RTS_STAGES.SIZE_TYPE(DA'size/STORAGE.UNIT),
        PRIO=> 1,
        NUM_ENTRIES=> 1,
        MASTER=> RTS_STAGES.CURRENT_MASTER,
        STATE=> (T1_PROC,COERCE(DA),NULL_DATA_AREA),
        LAST_CREATED=> NULL_TASK,
        ELABORATED=> ELABORATED,
        CREATED_TASK=> DA.T1);
     RTS_STAGES.COMPLETE_ACTIVATION;
     RTS_STAGES.ACTIVATE_TASKS(DA.T1);
     CHECK(13);
     -- make entry call: T1.E; #1
     RTS_RENDEZVOUS.CALL_SIMPLE(ACCEPTOR=>DA.T1,
                                E=> 1,
                                PARAMETER=>(NULL_DATA_AREA,0));
     -- make entry call: T1.E; #2
     RTS_RENDEZVOUS.CALL_SIMPLE(ACCEPTOR=>DA.T1,
                                E=> 1,
                                PARAMETER=>(NULL_DATA_AREA,0));
     --select T1.E else null; end select; #3
     RTS_RENDEZVOUS.CALL_CONDITIONAL(ACCEPTOR=>DA.T1,
                                     E=> 1,
                                     PARAMETER=>(NULL_DATA_AREA,0),
                                     RENDEZVOUS_SUCCESSFUL=> DA.RES);
     --select T1.E else null; end select; #4
     if DA.RES
     then CHECK(14); else CHECK(15); end if;
     RTS_RENDEZVOUS.CALL_CONDITIONAL(ACCEPTOR=>DA.T1,
                                     E=> 1,
                                     PARAMETER=>(NULL_DATA_AREA,0),
                                     RENDEZVOUS_SUCCESSFUL=> DA.RES);
     if DA.RES
     then CHECK(16); else CHECK(17); end if;
     -- make entry call: T1.STOP;
     RTS_RENDEZVOUS.CALL_SIMPLE(ACCEPTOR=>DA.T1,
                                E=> 2,
                                PARAMETER=>(NULL_DATA_AREA,0));

when T1_PROC=>
     RTS_STAGES.COMPLETE_ACTIVATION;
     CHECK(1);
     -- accept E; #1
     RTS_RENDEZVOUS.ACCEPT_CALL(E=> 1,
                                PARAMETER=> DA.PARAMETER);
     begin CHECK(2);
           RTS_RENDEZVOUS.COMPLETE_RENDEZVOUS;
     exception
       when others=> RTS_RENDEZVOUS.EXCEPTIONAL_COMPLETE_RENDEZVOUS
                     (RTS_EXCEPTIONS.CURRENT_EXCEPTION);
     end;
     CHECK(3);
     -- accept E; #2
     RTS_RENDEZVOUS.ACCEPT_CALL(E=> 1,
                                PARAMETER=> DA.PARAMETER);
     begin CHECK(4);
           RTS_RENDEZVOUS.COMPLETE_RENDEZVOUS;
     exception
       when others=> RTS_RENDEZVOUS.EXCEPTIONAL_COMPLETE_RENDEZVOUS
                     (RTS_EXCEPTIONS.CURRENT_EXCEPTION);
     end;
     CHECK(5);
     -- select accept E; #3
     --   else null;
     -- end select;
     RTS_RENDEZVOUS.SELECTIVE_WAIT(OPEN_ENTRIES=>(1..1=>1),
                                   D=> 0.0,
                                   SELECT_MODE=> RTS_RENDEZVOUS.ELSE_MODE,
                                   PARAMETER=> DA.PARAMETER,
                                   INDEX=> DA.RES2);
     case DA.RES2 is
     when 1=>
       begin
         CHECK(6);
         RTS_RENDEZVOUS.COMPLETE_RENDEZVOUS;
       exception
         when others=> RTS_RENDEZVOUS.EXCEPTIONAL_COMPLETE_RENDEZVOUS
                       (RTS_EXCEPTIONS.CURRENT_EXCEPTION);
       end;
       CHECK(7);
     when others=> CHECK(8);
     end case;
     -- select accept E;
     --   else null;
     -- end select;
     RTS_RENDEZVOUS.SELECTIVE_WAIT(OPEN_ENTRIES=>(1..1=>1),
                                   D=> 0.0,
                                   SELECT_MODE=> RTS_RENDEZVOUS.ELSE_MODE,
                                   PARAMETER=> DA.PARAMETER,
                                   INDEX=> DA.RES2);
     case DA.RES2 is
     when 1=>
       begin
          CHECK(9);
```

```
               RTS_RENDEZVOUS.COMPLETE_RENDEZVOUS;
             exception
               when others=> RTS_RENDEZVOUS.EXCEPTIONAL_COMPLETE_RENDEZVOUS
                           (RTS_EXCEPTIONS.CURRENT_EXCEPTION);
             end;
             CHECK(10);
           when others=> CHECK(11);
           end case;
           -- accept STOP;
           RTS_RENDEZVOUS.ACCEPT_CALL(E=> 2,
                                      PARAMETER=> DA.PARAMETER);
           begin CHECK(12);
                 RTS_RENDEZVOUS.COMPLETE_RENDEZVOUS;
           exception
             when others=> RTS_RENDEZVOUS.EXCEPTIONAL_COMPLETE_RENDEZVOUS
                           (RTS_EXCEPTIONS.CURRENT_EXCEPTION);
           end;
      when TERMINATE_PROC=>
           RTS_STAGES.COMPLETE_TASK;
      when TASKING_ERROR_PROC=>
           raise TASKING_ERROR;
      when others=>
           PUT("undefined PROC. ");
           PUT_LINE(PROCEDRUE'image(P));
           raise PROGRAM_ERROR;
      end case;
      exception when others=>
             ST.DATA:= DA.OLD_DATA;
             OLD(DA);
             raise;
      end;
      if IS_TASK(P) then COMPLETE; RTS_STAGES.COMPLETE_TASK; end if;
      ST.DATA:= DA.OLD_DATA;
      OLD(DA);
end CALL;

procedure DEALLOCATE(A: in out MACHINE.DATA_AREA) is
  AA: STORE:= COERCE(A);
  -- due to restriction RM 6.4.1(3) that actual of OLD must be a variable
  -- name or conversion, where COERCE is not recognized as a conversion.
begin OLD(AA); A:=NULL_DATA_AREA;
end DEALLOCATE;

begin INIT(RIGHT_TO_START_UP,AVAILABLE=> TRUE);
end PROCEDRUES;
```